

## CSCI 3081W Fall 2014 - Writing Assignment (Goal 2)

Name : Wen Chuan Lee

---

### Question 1: The First Principle of Software Testing

Assess how the tests we gave you performed against Bertrand Meyer's first Principle of Software Testing. Did the tests we gave you help reveal faults quickly? If the tests we gave you did help reveal faults quickly, what fault or faults did they help you reveal.

---

Bertrand Meyer states that the first Principle of Software Testing is "Definition", in which the author explains that "to test a program is to try to make it fail". This is evident in the test cases provided for all the iterations of this project, because the test cases given are meant to assess that the program is properly implemented and if there are faults that may exist in certain conditions. Throughout the iterations of developing a functional translator that would take a program written in a domain-specific language, the Forest Cover Analysis Language (FCAL) and return an equivalent C++ program by first scanning text, matching regular expressions (regex), creating a parse tree and returning a translation of the program, multiple test cases were provided to ensure correctness of the program in those cases. This shows that the test cases adhered to Meyer's first principle of software testing, as the test cases aim to make an erroneous program fail until a written implementation provides the correct (expected) assertions.

```
void test_parse_bad_syntax ( ) {  
    const char *text  
        = readInputFromFile ( "../samples/bad_syntax_good_tokens.dsl" ) ;  
    TS_ASSERT ( text ) ;  
    ParseResult pr = p->parse ( text ) ;  
    TS_ASSERT ( ! pr.ok ) ;  
}
```

**Figure 1a.** An example test case provided that tests for simple yet crucial functionality to ensure that the program correctly parses the linked list of tokens from an FCAL program that despite having bad syntax, should have correct tokens, which is asserted to be working before Iteration 3 which requires writing code to build an Abstract Syntax Tree.

The provided test cases also served to assist in revealing faults swiftly. The reason for this is that if there is an error in the program (that is detected by failing test cases), Cxx-Testing Suite will report this once all test cases have been executed, along with helpful information about which test case caused the failure by reporting which line contains the test case that failed.

```
wclee@athena-tux:~/3081repo/repo-group-VictoriousSecret/project/src$ make run-tests
./regex_tests
Running cxxtest tests (4 tests)....OK!
./scanner_tests
Running cxxtest tests (51 tests).....OK!
./parser_tests
Running cxxtest tests (10 tests).....
In ParserTestSuite::test_parse_mysample:
/home/wclee/3081repo/repo-group-VictoriousSecret/project/src/parser_tests.h:86:
Error: Assertion failed: text .
Failed 1 and Skipped 0 of 10 tests
Success rate: 90%
make: *** [run-tests] Error 1
```

**Figure 1b.** An example of the provided test cases revealing faults quickly. It can be seen that the test case at Line 86 failed an assertion where there was no text, allowing for faster error isolation.

A fault that was detected very quickly in developing Iteration 3 of the translator was the unparsing of the parsed FCAL program. The test cases were designed to test for some program correctness by first parsing a FCAL program, then storing the unparsed program (that is in FCAL) in a separate output file, and finally parsing that output again to ensure that the entire parsing process did not change the language of the program erroneously. This allowed for black-box testing in which white space and coding style that does not affect the language to be ignored, yet ensured proper syntax and grammar.

```
In AstTestSuite::test_sample_1:
/home/wclee/3081repo/repo-group-VictoriousSecret/project/src/ast_tests.h:62: Error: Test
failed: "sample_1.dsl failed to parse the first un-parsing."
/home/wclee/3081repo/repo-group-VictoriousSecret/project/src/ast_tests.h:62: Error:
Assertion failed: pr2.ok
```

**Figure 1c.** An example of Cxx-Test notifying which test failed, as well as where the failed assertion is located at. In this test case pr2.ok was flagged as false as there was a LexicalError token generated that could not be parsed properly to build the syntax tree. Upon manual review of the sample\_1.dsl file and corresponding unparse() methods, it was noticed that there was a typographical error in unparsing the program as FCAL where “let” was written as “Let”. Making the changes to the appropriate unparse() function then ensured that the test case passed successfully.

As indicated in the example above, the test cases were invaluable in revealing the faults in implementation very quickly, and is crucial in delivering a working program under the iterative development process. Therefore, the provided test cases revealed faults quickly and expedited the entire process of isolating errors made in programming, which in these examples are errors of having no text to parse as well as a typographical error that unparsed erroneously.

## Question 2: The Faults Unrevealed by Software Testing

Did your project have a fault or faults that the tests we gave you did not reveal? If so, what was the fault or faults? How did you find and fix the fault or faults?

---

“Program Testing, can be used to show the presence of bugs, but never to show their absence!” as famously quoted by Edsger W. Dijkstra on testing (also mentioned in Meyer’s Seven Principles of Software Testing), this is definitely true in the testing and development of the FCAL to C++ translator. There were two faults that the given tests did not reveal which are presented in chronological order. These were realized in testing the scanner in Iteration 1 and in the after Iteration 1 was completed. In development for Iteration 1, the fault was from writing the implementation of generating a linked-list of tokens which are returned by the scan() function and passed to the parser. While the test cases executed, none of them terminated successfully. Instead, there was an infinite loop caused by iterating through the linked list of nodes that did not crash the testing framework. This was followed by extensive code debugging and manual testing.

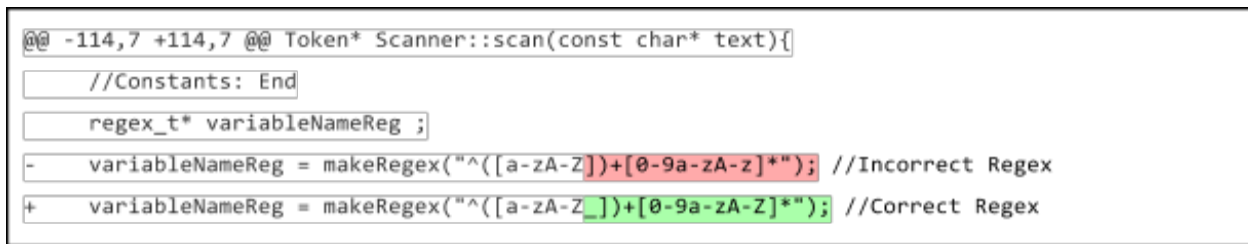
```
wclee@athena-tux:~/3081repo/repo-group-VictoriousSecret/project/src$ make run-tests
./regex_tests
Running cxxtest tests (4 tests)...OK!
./scanner_tests
Running cxxtest tests (40 tests).
```

**Figure 2a.** An example of Cxx-Test Suite executing, but never crashing or successfully completing. This was caused by the program infinitely iterating through the linked list of nodes when trying to assign a new node to the end of the list. The issue was resolved through manual testing and error isolation.

As displayed above, the test cases did not reveal the fault as Cxx-Test never terminated the running program and the program never crashed as well. This error was from an incorrect implementation of the linked-list in which memory was not allocated for a new Node and instead always pointed to the same Node object. The error was isolated through manual stepping of the code and fixed by having the program create a new Node object after pinpointing it. While this was a significant fault in the code written that could not be detected by the testing framework, it was fortunately resolved by manual debugging.

The second fault that was not revealed by the given test cases was in Iteration 1, in which the given parser framework was to be implemented with the scanner, which at the time passed all given regex and scanner tests. An error in writing the regex that matches variables had caused the parser tests to fail. The regular expression that matches variables would also incorrectly match the “[“ character, causing errors with the parsing when the translator ran on other FCAL programs. This fault was not revealed when using the provided test cases for Iteration 1. Instead it was only later (after passing all test cases in Iteration 1) in Iteration 2 was it realized when

additional test cases were written separately to investigate the cause for the parser to incorrectly parse matrix types (FCAL uses “[” and “]” to denote matrices).



```
@@ -114,7 +114,7 @@ Token* Scanner::scan(const char* text){
    //Constants: End
    regex_t* variableNameReg ;
-   variableNameReg = makeRegex("^([a-zA-Z])+([0-9a-zA-Z])*"); //Incorrect Regex
+   variableNameReg = makeRegex("^([a-zA-Z_])+([0-9a-zA-Z])*"); //Correct Regex
```

**Figure 2b.** A GitHub History of changes made to scanner.cpp, where the top highlighted portion denotes the incorrect regex that would match “[” and the bottom highlighted portion illustrating the correct regular expression that also takes into account for the ‘underscore’ character.

This fault was first found by writing additional test cases to match the matrix regex, and further investigation was done by writing a few lines of code that would print out the entire linked list of tokens and the matched characters to manually review where the error was. It was then fixed by making the changes as illustrated in Figure 2b and ensured to be functioning properly by writing even more regex test cases that extensively test for edge cases and matching of the square bracket character.

These two instances have shown that the test cases provided do not reveal all faults in the program, and they should not be expected to because more test cases were expected to be written, as test cases are only meant to test for errors that may exist in the code, not the lack of errors, as indicated in the First Principle of Software Testing.

---

### Question 3: Finding and Fixing Faults in Additional Testing

Give a concrete example of a test that you created to test your project that was effective at helping you find and fix a fault or faults. Include the test in your answer, describe what it does, and specify the fault or faults that it was intended to find. Did it find the fault or faults it was intended to find? Did it find other faults? Did it find any faults? Did writing the test help you avoid errors – and if so, how?

---

An example of a test case that was created to test the program in order to find and fix faults in the program aside from the regex tests was the test written for the parser to ensure that the FCAL program was parsed properly. While this was not a requirement explicitly stated in Iteration 2 and 3, the following test case was crucial in the development of the Abstract Syntax Tree (AST) as modifications to the parser in order to build the AST effects which tokens are matched. The test case is included on the next page.

```
void test_parse_simple() {  
    const char *text = ("main(){number = 1 + 2;}");  
    ParseResult pr = p->parse(text);  
    string msg("simple");  
    msg += "\n" + pr.errors;  
    TS_ASSERT(pr.ok);  
}
```

**Figure 3a.** The simple parser test used to ensure that the generated tokens were always parsed properly. Despite its simplicity in checking for a simple case, this test case was later a crucial test when developing code that builds the abstract syntax tree as it would indicate if the written code would cause incorrect matching of variables, parentheses, semicolons and integer types.

The test case was originally intended to ensure that the given parser (Iteration 2) was correctly implemented with the scanner from Iteration 1. This was achieved by first creating a string that contains a simple but syntactically correct FCAL program, which consisted of a main function that has variable *number* which saves the result of adding the integers 1 and 2. The parser, which would internally pass the string to the scanner and then use the list of tokens generated to generate a ParseResult, which stores the result of the parsing. As the black-box testing approach was used, the test case is not aware of the lists of tokens. The test case would then check if the parse result status was correct and that no parsing errors occurred.

The fault that the test case was intended to find was to check if the scanner functioned correctly with the given parser framework. During Iteration 2 when the test case was written, the test case did not manage to find the fault it was originally intended to find. However, this test case found other faults in Iteration 3, when the development for the code to build the AST began. Adhering to design principles, the code written to map the ParseResults to the AST was built by first ensuring a small part of the program functions as intended. This test case had a failing assertion initially as the code written to match the simple program caused a mismatch in the parsing by always matching one token ahead than expected.

Writing this test case (which highlighted a fault in the program that it was not intended to find) helped in avoiding errors as the isolated error was quickly fixed and the correct implementation of the code could be written for other, more complex parsings such as a long Matrix declaration that had multiple variables and token types. Hours of debugging all parts of the parser was prevented by writing this simple test case; a beauty of frequent software testing throughout the iterative development cycle.

Therefore, despite the test case not finding the fault it was intended to find, it found other faults in the early stages of development that definitely saved a lot of time and frustration, and must definitely be practiced in program design and development.

---

Reference: *Seven Principles of Software Testing*, Bertrand Meyer, August 2008

[http://www-users.cs.elslabs.umn.edu/classes/Fall-2014/csci3081/writing/PrinciplesOfSoftwareTesting\\_Meyer.pdf](http://www-users.cs.elslabs.umn.edu/classes/Fall-2014/csci3081/writing/PrinciplesOfSoftwareTesting_Meyer.pdf)